(4)

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY **AD-A218 613**	3. DISTRIBUTION/AVAILABILITY OF REPORT Unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-90-1088	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Cornell University / 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) Department of Computer Science Upson Hall, Cornell University Ithaca, NY 14853	7b. ADDRESS (City, State, and ZIP Code) 800 North Quincy St. Arlington, VA 22217-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Office of Naval Research / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N000014-86-K-0092
8c. ADDRESS (City, State, and ZIP Code) 800 North Quincy St. Arlington, VA 22217-5000	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
Priority Inversion and Its Prevention

12. PERSONAL AUTHOR(S)
Ozalp BABAOGLU, Keith MARZULLO, Fred B. SCHNEIDER

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM ______ TO ______	February 1990	27

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
real-time systems, priority scheduling, priority inversion

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A priority inversion occurs when a low-priority task causes execution of a higher-priority task to be delayed. The posibility of priority inversion complicates the analysis of systems that use priority-based schedulers because priority inversions invalidate the assumption that a task can be delayed only by higher-priority tasks. This paper formalizes priority inversion and gives sufficient conditions as well as some new protocols for preventing priority inversions.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
[✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Fred B. Schneider	(607) 255-9221	

Priority Inversion and Its Prevention*

Özalp Babaoğlu**
Keith Marzullo
Fred B. Schneider

TR 90-1088
February 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Babaoğlu and Marzullo are supported in part by the Department of Defense Advanced Research Projects Agency (DARPA) under grant N00140-87C-8904. Babaoğlu is also supported by the Commission of the European Communities under the ESPRIT Programme Basic Research Action Number 3092 (Predictably Dependable Computing Systems) and the Italian Ministry of Research and University. Schneider is supported in part by the Office of Naval Research under contract N00014-86-K-0092, the National Science Foundation under Grant No. CCR-8701103, and Digital Equipment Corporation. The views, opinions, and findings contained in this report are those of the authors and do not reflect the views of these agencies.

**Department of Mathematics, University of Bologna, Piazza Porta S. Donato 5, I-40127 Bologna, Italy.

Priority Inversion and its Prevention*

Özalp Babaoğlu† Keith Marzullo
Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

February 9, 1990

Abstract

A priority inversion occurs when a low-priority task causes execution of a higher-priority task to be delayed. The possibility of priority inversion complicates the analysis of systems that use priority-based schedulers because priority inversions invalidate the assumption that a task can be delayed only by higher-priority tasks. This paper formalizes priority inversion and gives sufficient conditions as well as some new protocols for preventing priority inversions.

*Babaoğlu and Marzullo are supported in part by the Department of Defense Advanced Research Projects Agency (DARPA) under grant N00140-87C-8904. Babaoğlu is also supported by the Commission of the European Communities under the ESPRIT Programme Basic Research Action Number 3092 (Predictably Dependable Computing Systems) and the Italian Ministry of Research and University. Schneider is supported in part by the Office of Naval Research under contract N00014-86-K-0092, the National Science Foundation under Grant No. CCR-8701103, and Digital Equipment Corporation. The views, opinions, and findings contained in this report are those of the authors and do not reflect the views of these agencies.

†Department of Mathematics, University of Bologna, Piazza Porta S. Donato 5, I-40127 Bologna, Italy.

1 Introduction

A *priority scheduler* controls access to a resource in accordance with some priority assignment. Each task is assigned a priority; whenever there is contention for a resource, access is granted to the task with the highest priority among those competing. Priority schedulers are frequently employed in real-time systems to allocate the processor among tasks that must produce results in a timely manner [6, 12].

A *priority inversion* occurs when a lower-priority task delays execution of a higher-priority task [5, 9]. For example, a task holding a write lock for some data will delay a task attempting to acquire a read lock for that data. If the holder of the write lock has a lower priority than the task attempting to acquire the read lock, then a lower-priority task is delaying a higher-priority task and a priority inversion has occurred.

The possibility of priority inversions creates difficulties for the designer of a real-time system. Not only must a task compete for resources with higher-priority tasks, but during a priority inversion, it loses resources to lower-priority ones. When a high-priority task τ_H is delayed by a lower-priority task τ_L, then τ_H effectively competes with all tasks assigned priorities at least that of τ_L, rather than with only those tasks assigned priorities at least that of τ_H. Since τ_L can be an arbitrary task, establishing that τ_H will meet a response-time goal requires reasoning involving all tasks in the system rather than just the subset with priority at least that of τ_H.

One approach to coping with priority inversion is to modify task priorities dynamically so that priority inversions are bounded and short in length [5, 10]. In these *priority inheritance* protocols, a task's priority is elevated to a level that is the maximum of its original priority and the priority of any task that is being delayed by it. Thus, priority inversions are permitted, but only in a carefully controlled way.

In this paper, we explore approaches to preventing priority inversion that do not involve modifying task priorities. In Sections 2 and 3, we formalize priority inversion and give sufficient conditions for its prevention. Based on these, some new protocols to prevent priority inversions from occurring are derived in Sections 4 and 5. The protocol of Section 4 is appropriate for systems where the times that tasks hold resources can be bounded; the protocol of Section 5 is appropriate for database systems, where tasks (transactions) can be aborted. In Section 6, we consider conditions for avoiding priority inversions in systems where there are multiple independent schedulers, each making allocation decisions for some subset of the resources. Section 7 puts our work in context and discusses some unsolved problems.

2 System Model

Formalizing priority inversion requires that we formalize the notions of priority assignment and delay. To do this, we model a system as a set of tasks $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ where a *task* is any computation that can be scheduled. Thus, our use of the term task is synonymous with alternatives such as *process*, *job*, and *transaction*.

2.1 Task Priorities

A *priority assignment* is an irreflexive, partial order[1] on T such that $\tau \succ \tau'$ whenever task τ has higher priority than τ'. Observe that this definition allows tasks to have incomparable priorities. Therefore, it is possible that neither $\tau \succ \tau'$ nor $\tau' \succ \tau$ holds for some pair of tasks τ and τ'. By assigning incomparable priorities to tasks, the number of constraints imposed by a priority assignment is reduced, avoiding the possibility of extraneous priority inversions.

Define the *peer group* of a task τ as the set of tasks τ' such that either $\tau' \succ \tau$ or τ' is incomparable to τ. In the absence of priority inversions, we need only consider τ and tasks that are in its peer group in analyzing whether τ will satisfy given response–time constraints. This is because only tasks in the peer group of τ can cause τ to be delayed.

2.2 Resources

Tasks can cause each other to be delayed in a variety of ways. Some of these are explicit, such as when one task awaits a message sent by another or when a lock held by one task prevents another from acquiring that lock. Other causes of delay are implicit. For example, the presence of finite–capacity, time-multiplexed resources, such as memory, processors, and I/O devices,

[1] An irreflexive partial order on a set is an asymmetric, irreflexive, transitive relation on pairs of elements from that set.

can lead to the (implicit) delay of a task requiring use of a resource by the task using that resource.

For our purposes we can abstract from these particulars, postulating that a system comprises a set of resources and a scheduler[2]. A task obtains access to a single unit of a resource r by invoking the

```
request(r)
```

operation and relinquishes that access by invoking the

```
release(r)
```

operation of the scheduler. Execution of a request operation delays the invoker until that request can be *granted.* Whether a request is granted is determined by the scheduler. A task may have only one outstanding request at a time. Thus, if multiple units of a resource or several different resources are needed for execution, a task must request and acquire them sequentially. We discuss the possibility of a task requesting multiple resources through a single operation in Section 7.3.

Request and release operations are not always explicitly invoked by tasks. Sometimes, these operations are invoked implicitly as part of some other system operation. For example, an operation to receive a message will implicitly invoke a request operation that is granted when the message becomes available for receipt. In other cases, request or release might not be invoked by tasks at all, instead being invoked due to other activity in the system. Consider a multiprogrammed processor that uses an interval timer to force task switches. An execution of the interval-timer interrupt handler can be regarded as (i) performing a release and a subsequent request for the task that was executing when the timer-interrupt occurred and then (ii) granting the request for the task that is next selected for execution on the processor.

Our request/release model turns out to be quite general. It can even be used to describe situations in which tasks are delayed because of some application-dependent aspect of the system state. For example, it is not unusual for a concurrent program to contain some form of conditional wait

[2]Most real systems have multiple schedulers, but postulating a single scheduler is not a limitation. It is always possible to model the effect of a collection of schedulers by using a single scheduler that makes allocation decisions using only information that would be available to the relevant scheduler.

statement that delays a task until the program variables satisfy some Boolean expression. (Most synchronization primitives are instances of conditional wait statements.) We can model such a conditional wait by regarding it as a request on a virtual resource. The request is granted if the Boolean expression is true; otherwise, the request is delayed. Execution of any assignment statement in another task that makes the Boolean expression true is treated as a release on the resource.

2.3 Delay

We model task delays by a binary relation. A task τ_i *waits for* task τ_j at time t, denoted $\tau_i \underset{t}{\rightarrow} \tau_j$, if and only if τ_i is delayed at time t in its request for some resource and τ_j can release that resource. Note that a task could be waiting for (any of) a set of tasks $\mathcal{T}$, each of which can release the resource being requested. The definition of $\underset{t}{\rightarrow}$ allows us to decompose this situation into a conjunction of waits-for relations:

$$\tau_i \underset{t}{\rightarrow} \mathcal{T} \equiv \bigwedge_{\tau \in \mathcal{T}} (\tau_i \underset{t}{\rightarrow} \tau). \tag{1}$$

We assume that a grant/delay decision can be made at the time of a request and that a delay always can be attributed to some set $\mathcal{T}$ of tasks. For example, if there are multiple units of the resource available, the delay is attributed to the set of all tasks that have been granted but not yet released the resource. And, in the case of a request for a virtual resource associated with a conditional wait, the set $\mathcal{T}$ contains all those tasks that can execute assignment statements that make true the Boolean condition being awaited.

3 Characterizing Priority Inversion

In order to characterize priority inversion, we must reason about the transitive closure of the waits-for relation. We say that a task τ_i *implicitly waits for* task τ_j at time t, denoted $\tau_i \underset{t}{\leadsto} \tau_j$, if and only if either $\tau_i \underset{t}{\rightarrow} \tau_j$ or there exists τ_k such that $(\tau_i \underset{t}{\leadsto} \tau_k) \wedge (\tau_k \underset{t}{\rightarrow} \tau_j)$.

Priority inversion has occurred at time t when progress of a task is blocked by the actions of a lower priority task. Thus, a system contains a prior-

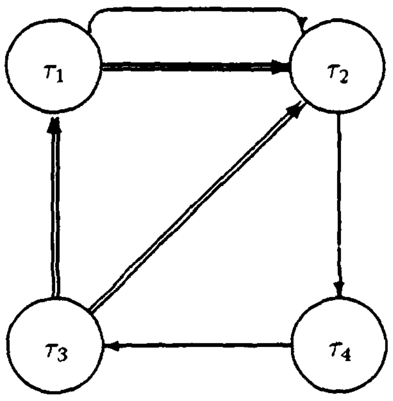

Figure 1: Four–Task Composite System Graph.

ity inversion at time t if and only if there exist tasks τ_i and τ_j such that $(\tau_i \underset{t}{\leadsto} \tau_j) \wedge (\tau_i \succ \tau_j)$.

It will be convenient to represent the priority assignment and waits–for relations in effect at a given time t as graphs. The directed graph corresponding to a priority assignment $\succ$ on a set of tasks T is $P = (T, E_P)$ where $E_P = \{(u,v) \mid v \succ u\}$. Thus, the nodes of P represent tasks and an edge is drawn from a task to all higher priority tasks. Similarly, the waits–for relation at time t can be represented as the directed graph $W = (T, E_W)$ where $E_W = \{(u,v) \mid u \underset{t}{\rightarrow} v\}$. Here, edges are drawn from a task to all other tasks that it waits for.

Given these graphs, the system state at a time t can be represented by a *composite system graph*, $G = (T, E_P \cup E_W)$. Figure 1 depicts such a graph for a system of four tasks. Single–arrow edges represent waits–for relations and double–arrow edges represent priority relations. Thus, the graph depicts the situation where there are four tasks such that $\tau_2 \succ \tau_1 \succ \tau_3$ and $(\tau_1 \underset{t}{\rightarrow} \tau_2)$, $(\tau_2 \underset{t}{\rightarrow} \tau_4)$, $(\tau_4 \underset{t}{\rightarrow} \tau_3)$. Note that there can be multiple edges between a pair of nodes.

The following theorem uses a composite system graph to characterize the existence of a priority inversion at time t. It is based on directed cycles containing exactly one priority edge. We call such cycles π–cycles.

Theorem 1 *A composite system graph G describes a priority inversion if and only if G contains a directed cycle involving exactly one priority edge (π-cycle).*

Proof: Without loss of generality, let $\tau_j \Rightarrow \tau_i \rightarrow \cdots \rightarrow \tau_j$ be a π-cycle of G. By definition of $\rightsquigarrow$, we have $\tau_i \underset{t}{\rightsquigarrow} \tau_j$. From priority edge $\tau_j \Rightarrow \tau_i$, we have the relation $\tau_i \succ \tau_j$. Thus, the system contains a priority inversion according to the definition given above. The result that a priority inversion implies a π-cycle in G follows trivially from the construction of a composite system graph. □

The system depicted in Figure 1 contains two priority inversions, corresponding to the two π-cycles ($\tau_3 \Rightarrow \tau_1 \rightarrow \tau_2 \rightarrow \tau_4 \rightarrow \tau_3$ and $\tau_3 \Rightarrow \tau_2 \rightarrow \tau_4 \rightarrow \tau_3$). Task τ_3 is responsible for delaying tasks τ_1, τ_2, and τ_4 and has lower priority than τ_1 and τ_2.

From Theorem 1, we conclude that any condition that prevents a composite system graph G from having a π-cycle is a sufficient condition for avoiding priority inversions. Examples of such conditions are the following:

1. *No Priority Assignment.* If all tasks were incomparable, then the priority graph would be empty and a π-cycle could never form.

2. *No Delay.* Without delay, the waits-for graph would be empty, guaranteeing the absence of the π-cycles.

3. *Preemption.* A waits-for relation persists until a task relinquishes a resource. Preemption causes a task to relinquish a resource, so preemption can change the waits-for relation in a way that prevents a π-cycle from forming.

4. *No π-Cycle.* The waits-for and the priority relations must form in a particular fashion for priority inversion to exist. Theorem 1 establishes the relevant property as a π-cycle.

Note that preemption both removes and adds elements to the waits-for relation. By preempting a resource r that had been granted to a task τ_i and granting r to τ_j, all waits-for relations from τ_j to other tasks are removed and waits-for relations from τ_i to all tasks that have access to the resource are added.

Any strategy for preventing priority inversions ultimately must be based on avoiding the π-cycle condition of Theorem 1. In the strategies that follow, we do just this by ensuring that at least one of the sufficient conditions above holds. First, in Section 4, we show how by eliminating the possibility of certain waits-for relations, a π-cycle is avoided. Thus, this strategy is based on condition 4, No π-Cycle. Then, in Section 5, we show an application of preemption by giving a timestamp-based concurrency controller. This strategy is based on condition 3, Preemption.

4 Applying the Theory: Developing Reservation Protocols

The π-cycle condition of Theorem 1 can be prevented by having each task reserve in advance the interval during which it will hold a resource and using that information to prevent certain waits-for relations from forming. If reservations from a low-priority task never overlap with reservations from a higher-priority task, then priority inversions become impossible. In this section, we develop protocols that exploit this insight for avoiding priority inversions. The method by which we obtain these protocols is more important than the protocols themselves. Deriving the protocols provides an opportunity for us to show how our theory can be used to obtain policies for avoiding priority inversions from conditions that ensure π-cycles are impossible.

Let $H_i = \{\tau \mid \tau \succ \tau_i\}$ be the set of tasks that have higher priority than τ_i, and let $I_i = \{\tau \mid \neg(\tau \succ \tau_i) \wedge \neg(\tau_i \succ \tau)\}$ be the set of tasks incomparable to τ_i. Thus, the peer group for a task τ_i is $PG_i = H_i \cup I_i$. For each resource r, assume that each task τ_i is able to compute $hold_i^r(t)$, the upper bound on the amount of time that τ_i will hold r the next time (with respect to t) it is granted r, and $next_i^r(t)$, the lower bound on the next time (with respect to t) τ_i will request r. During the interval between when task τ_i requests resource r and when it releases r, define $next_i^r(t)$ to equal t. And, if τ_i holds r at time t then define $hold_i^r(t)$ to be an upper bound on the remaining amount of time τ_i will hold r. The reservation protocols we derive require that at times of allocation decisions, the scheduler be able to interrogate a set of tasks for their $next_i^r(t)$ values for a set of resources. We are assuming that the communication delays between tasks and the scheduler are negligible with respect to $hold_i^r(t)$ and $next_i^r(t)$. In case they are not, the allocation policies can be easily modified to account for them.[3]

An allocation policy can be derived from any program invariant[4] that precludes formation of π-cycles. Not only must the program invariant imply that there is no π-cycle present in the current state, but also that the current state is not one from which formation of a π-cycle is inevitable. Therefore, it suffices that the program invariant imply the stronger condition that there is no π-cycle present in the current state and that the current state is not one from which formation of a π-cycle is *possible*. Although such a stronger program invariant could rule out safe states, it is usually easier to construct and maintain.

In order to construct a program invariant that rules out the possibility of future π-cycles, define the predicate $\tau_j \overset{?t'}{\underset{t}{\rightarrow}} \tau_i$ to mean: given the resources τ_i has allocated at time t, it is possible $\tau_j \underset{t'}{\rightarrow} \tau_i$ will hold for some t' such that $t' \geq t$. Thus, letting $R_i(t)$ be the set of resources that τ_i has allocated at time t,

$$\tau_j \overset{?t'}{\underset{t}{\rightarrow}} \tau_i \overset{\text{def}}{=}$$
$$(t' \geq t) \wedge (\exists r\colon\ r \in R_i(t)\colon\ (hold_i^r(t) + t > t') \wedge (next_j^r(t) \leq t')).$$

Note that if $\tau_j \underset{t}{\rightarrow} \tau_i$, then at time t there is some resource $r \in R_i(t)$ and r has been requested by τ_j. Therefore, if $\tau_j \underset{t}{\rightarrow} \tau_i$, then, by definition, $hold_i^r(t) > 0$ and $next_j^r(t) = t$, and so:

$$(\tau_j \underset{t}{\rightarrow} \tau_i) = (\tau_j \overset{?t}{\underset{t}{\rightarrow}} \tau_i) \qquad (2)$$

4.1 Policy 1

In a system where resources are shared infrequently, the most common π-cycle would involve just two tasks τ_i and τ_j (say) such that $\tau_j \underset{t}{\rightarrow} \tau_i \wedge \tau_j \succ \tau_i$. An obvious program invariant to choose is the negation of this predicate:[5]

[3]In any implementation, $hold_i^r(t)$ and $next_i^r(t)$ would need only be computed on-demand at times of resource allocation. They can be based on empirical data collected during previous executions or on an *a priori* analysis of the code that is being executed.

[4]A *program invariant* is an assertion about the program state that is not invalidated by program execution.

[5]We write $A \supset B$ to denote the logical implication, A implies B.

$$(\forall \tau_i, \tau_j:\ \tau_j \underset{t}{\rightarrow} \tau_i \supset \neg(\tau_j \succ \tau_i)) \tag{3}$$

However, (3) does not imply that there can be no π-cycle in the composite system graph. The following predicate does:

$$(\forall \tau_i, \tau_j:\ \tau_j \underset{t}{\rightsquigarrow} \tau_i \supset \neg(\tau_j \succ \tau_i)) \tag{4}$$

So, we strengthen (3). First, note that since $\succ$ is asymmetric, we have:

$$(\forall \tau_i, \tau_j:\ \tau_i \succ \tau_j \supset \neg(\tau_j \succ \tau_i)) \tag{5}$$

Thus,

$$(\forall \tau_i, \tau_j:\ \tau_j \underset{t}{\rightarrow} \tau_i \supset \tau_i \succ \tau_j) \tag{6}$$

implies (3), and so if (6) also implied (4) then (6) would imply there can be no π-cycle in the composite system graph. We now show that (6) implies (4).

Assume (6) holds. Since $\succ$ is a transitive relation, we have from (6) and the definition of $\underset{t}{\rightsquigarrow}$ that:

$$(\forall \tau_i, \tau_j:\ \tau_j \underset{t}{\rightsquigarrow} \tau_i \supset \tau_i \succ \tau_j)$$

The last equation together with (5) imply (4). Thus, (6) implies that there can be no π-cycle in the composite system graph and can be used in constructing the program invariant.

Replacing $\tau_j \underset{t}{\rightarrow} \tau_i$ according to (2) we get:

$$(\forall \tau_i, \tau_j:\ \tau_j \overset{?t}{\underset{t}{\rightarrow}} \tau_i \supset \tau_i \succ \tau_j) \tag{7}$$

Unfortunately, (7) does not imply that the current state is one from which later formation of π–cycles is impossible. For example, suppose a task τ_i can allocate a resource r that a task τ_j will later request and $\tau_j \succ \tau_i$ holds. If τ_i allocates r, then (7) is not invalidated. However, if τ_j subsequently attempts to allocate r, then (7) is invalidated because the request cannot be granted (without preempting r from τ_i). So, (7) was not strong enough to prevent subsequent formation of a π-cycle.

We can strengthen (7) by weakening its antecedent:

$$(\forall \tau_i, \tau_j\colon\ (\exists t'\colon\ t \le t'\colon\ \tau_j \overset{?t'}{\underset{t}{\rightarrow}} \tau_i)\ \supset\ \tau_i \succ \tau_j)$$

By construction, this does prevent the possibility of π-cycles forming in subsequent states. Taking the contrapositive, substituting for the definition of $\overset{?t'}{\underset{t}{\rightarrow}}$, and performing some algebraic manipulations in order to eliminate t', results in the following strengthening:

$$\mathcal{I}_1\colon\quad (\forall \tau_i, \tau_j\colon\ \neg(\tau_i \succ \tau_j)\ \supset\ (\forall r\colon\ r \in R_i(t)\colon\ hold_i^r(t) + t \le next_j^r(t)))$$

Thus, we can use $\mathcal{I}_1$ as a program invariant for ensuring that π-cycles cannot form during execution.

To ensure that $\mathcal{I}_1$ holds throughout execution, we must show that it is true initially and that execution does not invalidate it. Assuming that tasks are initially started with no resources allocated to them, $R_i(0)$ will be empty for all tasks τ_i, and so $\mathcal{I}_1$ is initially true. To ensure that $\mathcal{I}_1$ is not invalidated by execution, assume that it is true and some task τ_i requests a resource r'. Let $ok(r', \mathcal{I}_1)$ denote those system states in which r' could be added to $R_i(t)$. Then, any policy that ensures a condition $C(r', \tau_i)$ holds before r' is granted to τ_i, such that

$$C(r', \tau_i) \wedge \mathcal{I}_1\ \supset\ ok(r', \mathcal{I}_1)$$

is valid, will ensure that $\mathcal{I}_1$ holds throughout execution (hence the formation of π-cycles is precluded).

Using the definition in [4] of wp for assignment statement $R_i(t) := R_i(t) \cup \{r'\}$, we compute that

$$ok(r', \mathcal{I}_1)\ =\ wp(R_i(t) := R_i(t) \cup \{r'\}, \mathcal{I}_1).$$

Thus, for

$$C(r', \tau_i) \overset{\text{def}}{=} (\forall \tau_j\colon\ \neg(\tau_i \succ \tau_j)\ \supset\ hold_i^{r'}(t) + t \le next_j^{r'}(t))$$

we have

$$C(r', \tau_i) \wedge \mathcal{I}_1 \quad \supset \quad ok(r', \mathcal{I}_1).$$

because

$$C(r', \tau_i) \wedge \mathcal{I}_1\ =\ wp(R_i(t) := R_i(t) \cup \{r'\}, \mathcal{I}_1).$$

Therefore, provided $C(r', \tau_i)$ holds before r' is granted to τ_i we can conclude that $\mathcal{I}_1$ is not invalidated by program execution. This can be done by ensuring that the scheduler never grants r' to τ_i unless $C(r', \tau_i)$ holds. Rearranging terms in the definition for $C(r', \tau_i)$ results in the following (equivalent) policy:

Policy 1 *The request of task τ_i for resource r' at time t is granted if*

$$hold_i^{r'}(t) \le \min_{\tau_j \in PG_i} (next_j^{r'}(t) - t).$$

Otherwise, the request is delayed.

In words, a task is granted a resource r' only if it will release r' before any task in its peer group requests r'. Note that the incomparable tasks must be involved in the allocation decision—it is necessary that τ_j range over I_i as well as H_i.

Policy 1 is rather conservative. It does not allow waits-for edges to develop between incomparable tasks in fear of a cycle developing indirectly between two comparable tasks. However, if the priority assignment is a total order and $PG_i = H_i$, then a reservation is made with respect to all tasks with higher priority, just as we would expect.

4.2 Policy 2

Another predicate that implies there can be no π–cycle in the composite system graph is the following:

$$(\forall \tau_i, \tau_j\colon\ \tau_j \succ \tau_i \ \supset\ \neg(\exists \tau_k, \tau_\ell\colon\ \tau_\ell \underset{t}{\rightarrow} \tau_i \wedge \tau_j \underset{t}{\rightarrow} \tau_k)) \tag{8}$$

This is because all π-cycles contain an instance of Figure 2. Rewriting (8) and replacing $\tau_j \underset{t}{\rightarrow} \tau_i$ according to (2) we get:

$$(\forall \tau_i, \tau_j\colon\ \tau_j \succ \tau_i \ \supset\ \neg(\exists \tau_k, \tau_\ell\colon\ \tau_\ell \underset{t}{\overset{?t}{\rightarrow}} \tau_i \wedge \tau_j \underset{t}{\overset{?t}{\rightarrow}} \tau_k)) \tag{9}$$

Although this predicate ensures that no π–cycle exists in the current state, it does not imply that the current state is one from which formation of π-cycles is impossible. As in the derivation of Policy 1, it is not difficult to construct a scenario where (9) is preserved up until the point when a state is reached where a π-cycle is inevitable. And, as before, the problem is solved

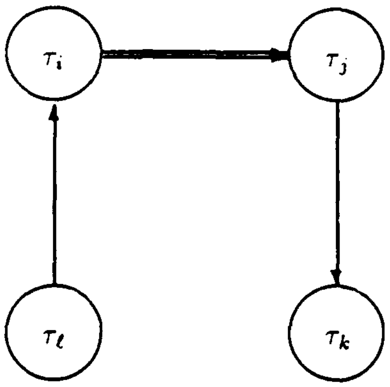

Figure 2: Bad Composite System Graph.

by strengthening. In particular, observe that $\neg(\exists \tau_\ell, t_2\colon\ t \le t_2\colon\ \tau_\ell \overset{?t_2}{\underset{t}{\rightarrow}} \tau_i)$ implies $\neg(\exists \tau_\ell\colon\ \tau_\ell \overset{?t}{\underset{t}{\rightarrow}} \tau_i)$ and that $\neg(\exists \tau_k, t_1, t_2\colon\ t \le t_1 \le t_2\colon\ \tau_j \overset{?t_2}{\underset{t_1}{\rightarrow}} \tau_k)$ implies $\neg(\exists \tau_k\colon\ \tau_j \overset{?t}{\underset{t}{\rightarrow}} \tau_k)$. We can use these facts to strengthen (9) by strengthening its consequent:

$$\begin{aligned}&(\forall \tau_i, \tau_j\colon\ \tau_j \succ \tau_i \supset\\ &\quad \neg(\exists \tau_k, \tau_\ell, t_1, t_2\colon\ t \le t_1 \le t_2\colon\ \tau_\ell \overset{?t_2}{\underset{t}{\rightarrow}} \tau_i \ \wedge\ \tau_j \overset{?t_2}{\underset{t_1}{\rightarrow}} \tau_k))\end{aligned} \tag{10}$$

By using two separate times t_1 and t_2, we characterize scenarios where τ_j is delayed due to a later allocation by τ_k.

Substituting in (10) according to the definition of $\overset{?t'}{\underset{t}{\rightarrow}}$, we obtain:

$$\begin{aligned}&(\forall \tau_i, \tau_j\colon\ \tau_j \succ \tau_i \supset\\ &\quad \neg(\exists \tau_k, \tau_\ell, t_1, t_2, r_i, r_k\colon\ t \le t_1 \le t_2 \ \wedge\ r_i \in R_i(t) \ \wedge\ r_k \in R_k(t_1)\colon\\ &\qquad hold_i^{r_i}(t) + t > t_2 \ \wedge\ t_2 \ge next_\ell^{r_i}(t) \wedge\\ &\qquad hold_k^{r_k}(t_1) + t_1 > t_2 \ \wedge\ t_2 \ge next_j^{r_k}(t_1)))\end{aligned}$$

Using algebra to eliminate variable t_2 and moving the negation inside the quantification, results in the following strengthening:

$$(\forall \tau_i, \tau_j\colon\ \tau_j \succ \tau_i \supset$$

$$(\forall \tau_k, \tau_\ell, t_1, r_i, r_k\colon\ t \le t_1 \ \wedge\ r_i \in R_i(t) \ \wedge\ r_k \in R_k(t_1)\colon$$
$$hold_i^{r_i}(t) + t \le next_\ell^{r_i}(t) \ \vee\ hold_k^{r_k}(t_1) + t_1 \le next_j^{r_k}(t_1) \ \vee$$
$$hold_i^{r_i}(t) + t \le next_j^{r_k}(t_1) \ \vee\ hold_k^{r_k}(t_1) + t_1 \le next_\ell^{r_i}(t)))$$

Deleting three of the disjuncts in the consequent then results in the following strengthening.

$$(\forall \tau_i, \tau_j\colon\ \tau_j \succ \tau_i \ \supset$$
$$(\forall \tau_k, t_1, r_i, r_k\colon\ t \le t_1 \ \wedge\ r_i \in R_i(t) \ \wedge\ r_k \in R_k(t_1)\colon$$
$$hold_i^{r_i}(t) + t \le next_j^{r_k}(t_1)))$$

Any subset of the disjuncts could have been deleted, with other choices leading to different policies.

We can further simplify by removing references to τ_k and t_1. We do this by strengthening based on the following observations. First, because $R_k(t) \subseteq R$ holds, replacing references to $R_k(t)$ by R results in a stronger consequent. Second, because $t \le t_1$, we conclude that $next_j^{r_k}(t) \le next_j^{r_k}(t_1)$, and so replacing t_1 in $hold_i^{r_i}(t) + t \le next_j^{r_k}(t_1)$ by t results in a stronger consequent. We, therefore, obtain:

$$\mathcal{I}_2\colon\ (\forall \tau_i, \tau_j\colon\ \tau_j \succ \tau_i \ \supset$$
$$(\forall r_i, r_k\colon\ r_i \in R_i(t) \ \wedge\ r_k \in R\colon\ hold_i^{r_i}(t) + t \le next_j^{r_k}(t)))$$

To ensure that $\mathcal{I}_2$ holds throughout execution, we show that it is true initially and that execution does not invalidate it. Assuming that tasks are initially started with no resources allocated to them, $R_i(0)$ will be empty for all tasks τ_i, and so $\mathcal{I}_2$ is initially true. To ensure that $\mathcal{I}_2$ is not invalidated by execution, assume that it is true and some task τ_i requests a resource r'. We desire a condition $C(r', \tau_i)$ such that

$$C(r', \tau_i) \wedge \mathcal{I}_2 \ \supset\ ok(r', \mathcal{I}_2)$$

is valid. Again, using the definition of wp for assignment statement $R_i(t) := R_i(t) \cup \{r'\}$, we can verify that any choice for $C(r', \tau_i)$ must imply:

$$(\forall \tau_j, r_k\colon\ r_k \in R\colon\ \tau_j \succ \tau_i \ \supset\ hold_i^{r'}(t) + t \le next_j^{r_k}(t))$$

Rearranging terms in the definition for $C(r', \tau_i)$ results in the following (equivalent) policy:

Policy 2 *Let R be the set of all shared resources. The request of task τ_i for resource r' is granted if*

$$hold_i^{r'}(t) \leq \min_{\tau_j \in H_i}(\min_{r_k \in R}(next_j^{r_k}(t) - t)).$$

Otherwise, the request is delayed.

Thus, a task is granted a resource r' only if it will release r' before any higher–priority task requests *any* resource.

One way Policies 1 and 2 can be compared is by considering the composite systems graphs one policy allows and the other policy avoids. In a system with a total order priority assignment, Policy 1 is superior, while in a system where there are many incomparable tasks and few shared resources, Policy 2 is appropriate. Depending on the application, there may be other properties of the possible composite systems graphs that can be exploited in order to derive a better resource allocation policy.

5 Applying the Theory: Developing a Concurrency Control Protocol

We now consider a strategy in which the π–cycle condition of Theorem 1 can be prevented in a database system by designing a timestamp–based concurrency controller that prevents priority inversions. Other database concurrency controllers that avoid priority inversions use locking [1, 10]. Thus, by applying our theory, we have been able to derive the first timestamp–based concurrency controller for avoiding priority inversions.

5.1 Serializability and Priority Inversion

A task accesses a database by encapsulating its reads and writes on the database within a transaction. A concurrency controller schedules these transactions so that their execution is *serializable*—that is, each transaction either commits or aborts, and execution of the committed transactions is equivalent to executing them in some serial order. A transaction τ_i *precedes* a transaction τ_j in a history h, written $\tau_i \ll \tau_j$, if in all serial executions equivalent to h, τ_i executes before τ_j. Serializability of a set of transactions T

can be characterized in terms of a *serialization graph*, $G = (T, E_C)$ where $E_C = \{(u, v) \mid u \ll v\}$. By definition, $\ll$ is an irreflexive partial order, so G is acyclic.

By allowing only serializable executions, a concurrency controller ensures that the serialization graph G never contains cycles [2]. For example, suppose some operation α_i of τ_i was executed before a conflicting operation β_j of τ_j, where two operations *conflict* if their execution order cannot be interchanged without altering the effects of one or the other. In this case, a concurrency controller effectively adds an edge from τ_i to τ_j in G. If two more conflicting operations, γ_i of τ_i and δ_j of τ_j, are to be executed, then γ_i must execute before δ_j, since to do otherwise would add an edge in G from τ_j to τ_i, creating a cycle.

Delays

One way to ensure that a serialization graph never contains cycles is by delaying execution of transactions. Henceforth, we will assume that if $\tau_i \underset{t}{\rightarrow} \tau_j$ due to a delay introduced by a concurrency controller, then $\tau_j \ll \tau_i$. This is a reasonable assumption because were it not true, then there would exist an equivalent serial execution in which τ_i precedes τ_j yet the concurrency controller delays some operation α_i of τ_i until operation β_j of τ_j completes. Such a delay would be capricious, since all the operations of τ_i could execute before any operation of τ_j and yield the same result as when α_i is delayed. It is, therefore, not surprising that all the concurrency control algorithms that we know of satisfy this assumption.

By delaying transactions, a concurrency controller can cause priority inversions. Define a *priority-ordered* concurrency controller to be one that ensures

POCC$_1$: For all transactions τ_i and τ_j, if τ_j starts before τ_i completes and $\tau_j \succ \tau_i$, then $\tau_j \ll \tau_i$.

POCC$_1$ ensures that higher-priority transactions are ordered before concurrently executed lower-priority ones.

We now show that a priority-ordered concurrency controller cannot introduce priority inversions: By assumption, delay edges that are in a composite system graph and can be attributed to the concurrency controller have corresponding conflict edges in the serialization graph. By POCC$_1$, for every priority edge in the composite system graph there is a corresponding conflict edge in the serialization graph. Thus, there is a π-cycle in the composite system graph only if there is a cycle in the serialization graph. Since a concurrency controller ensures the absence of cycles in the serialization graph, there can be no π-cycle in the composite system graph.

Aborts

In addition to delaying transactions, some concurrency controllers avoid cycles in the serialization graph by aborting transactions. Aborting a transaction can cause a priority inversion—for example, when a lower-priority transaction causes a higher-priority transaction to abort. In terms of our model, aborting a transaction can be regarded as having it wait for some (virtual) resource held by other transactions. This is sensible because a transaction τ_a that is aborted by the concurrency controller and later rescheduled does no useful work prior to its restart, and, in our model, a transaction that can do no useful work is considered blocked.

Whether or not aborting τ_a will cause a priority inversion depends on the priorities of transactions holding the virtual resource that is being requested by (and is blocking) τ_a. The set B_a of transactions holding this virtual resource are those that, had they not executed at all, would not have led to τ_a being aborted. Thus, if each transaction in B_a is in the peer group of τ_a, then aborting τ_a to avoid a cycle in the serialization graph does not create a priority inversion. Let C be the set of transactions involved in cycles in the serialization graph, and let A be the subset of C that are aborted in order to remove those cycles. Then, $B_a \subset C - A$ holds, and we have:

POCC$_2$: A concurrency controller that aborts a transaction τ_a will avoid priority inversions provided B_a is a subset of the peer group of τ_a.

5.2 Timestamp-based Concurrency Controllers

Timestamp-based concurrency controllers work by assigning a unique *timestamp* $ts(\tau_i)$ to each transaction τ_i. These timestamps are used to totally order transactions. The ordering is such that if $ts(\tau_i) < ts(\tau_j)$, then there is an edge in the serialization graph from τ_i to τ_j.

Timestamp-based concurrency controllers both delay transactions and abort transactions. In order to ensure that such a concurrency controller does not introduce priority inversions, two conditions suffice. The first condition is that the concurrency controller be priority–ordered, since being priority-ordered implies that delays will not introduce priority inversions. The second is that aborts do not introduce priority inversions. We now consider each of these conditions in detail.

A timestamp-based concurrency controller can be made priority–ordered by suitable assignment of timestamps. This is because POCC$_1$ requires that certain edges exist in a serialization graph. Since the concurrency controller adds such edges by assigning timestamps, it suffices to ensure that

POTS: If τ_j starts before τ_i completes and $\tau_j \succ \tau_i$, then $ts(\tau_j) < ts(\tau_i)$.

It is not hard to assign such timestamps. For simplicity, assume integer priorities; extension to general priorities is straightforward. Also assume there exist P priority levels $1, 2, \ldots, P$, where level $\ell_i \succ \ell_j$ if and only if $\ell_i > \ell_j$, and assume that timestamps are real numbers. Let

max^c	be the largest timestamp of all committed transactions,
min_i^a	be the smallest timestamp of all active transactions of priority i, and
max_i^a	be the largest timestamp of all active transactions of priority i.

If no transactions of priority i are active, then the value of min_i^a and max_i^a is defined to be $\bot$, where $\min(x, \bot) = \max(x, \bot) = x$. Initially, $max^c = 0.0$ and for all priority levels ℓ, $min_\ell^a = max_\ell^a = \bot$. A timestamp s for a new transaction with priority p can be computed by finding upper and lower bounds for its value and selecting a unique value in that interval. To be able to commit, the value of s must be larger than max^c; to satisfy POTS, it must be larger than the timestamps assigned to all transactions with higher-priority and smaller than the timestamps assigned to all transactions with lower priority. This is implemented by the code in Figure 3.

Having ensured that the concurrency controller is priority–ordered, it only remains to ensure that aborts do not introduce priority inversions. Suppose operation α_i from transaction τ_i is submitted before a conflicting operation β_j from τ_j. If $ts(\tau_i) > ts(\tau_j)$, then executing these operations in the order

```
$low := \mathbf{max}[max^c, max^q_\ell\colon\ 1 \le \ell \le p]$;
$high := \mathbf{min}[min^q_\ell\colon\ p < \ell \le P]$;
if $(high = \perp)$ then $s := low + 1$
                    else $s := (low + high)/2$;
$min^q_\ell := \mathbf{min}[min^q_\ell, s]$;
$max^q_\ell := \mathbf{max}[max^q_\ell, s]$;
```

Figure 3: Timestamp Allocator

they are submitted would create a cycle in the serialization graph[6]. Thus, the concurrency controller must abort either τ_i or τ_j to avoid this cycle. From POTS, if $\tau_j \succ \tau_i$ then $ts(\tau_j) < ts(\tau_i)$ holds, so, according to POCC$_2$, no priority inversion can result by aborting $ts(\tau_i)$. The following rule, therefore, describes a rule for aborting transactions without introducing priority inversions.

Priority Abort Rule: If α_i from transaction τ_i is submitted before conflicting operation β_j from τ_j and $ts(\tau_i) > ts(\tau_j)$, then τ_i is aborted to avoid a cycle in the serialization graph.

Notice that this rule is the opposite of what is traditionally used in timestamp-based concurrency controllers [2]. Traditionally, the transaction that submitted the last operation (e.g. τ_j above) would be aborted because this eliminates all cycles introduced by that operation. For example, suppose that $ts(\tau_i) = i$ and the following sequence of operations are submitted, where $r_i[x]$ denotes an operation by transaction τ_i to read x and $w_i[x]$ denotes an operation by transaction τ_i to write x:

$r_2[x]\ r_3[x]\ w_1[x]$.

Performing $w_1[x]$ would introduce two cycles in the serialization graph—one involving τ_1 and τ_2, the other involving τ_1 and τ_3. The traditional abort rule would abort τ_1, but would create a priority inversion if $\tau_1 \succ \tau_2$ or $\tau_1 \succ \tau_3$. The Priority Abort Rule would abort both τ_2 and τ_3 and cannot cause a priority inversion because of the way timestamps are assigned.

[6]Actually, if both operations are writes, the second write can be ignored and no conflict occurs [11].

Implementing the Priority Abort Rule is not completely straightforward. See [7] for a detailed explanation of such an implementation.

6 Systems with Multiple Schedulers

It is not uncommon for system resources to be partitioned into disjoint subsets, where each subset is controlled by an independent scheduler. For example, it is typical for the processors of a system to be scheduled by a CPU scheduler, disk drives to be scheduled by some device driver module, and database accesses to be scheduled by the concurrency control module of a database manager. A database transaction would interact with all three separate schedulers during its execution.

Although using multiple, independent schedulers simplifies construction of a system, it complicates the avoidance of priority inversions. This is because a scheduler's decision to delay granting some resource to a task must be made using only partial information about the system state, and a delay caused by one scheduler might cause a priority inversion with tasks being delayed by other schedulers. To see this, consider a system with schedulers S_1 and S_2 and tasks τ_i, τ_j, and τ_k, where $\tau_i \succ \tau_k$. Further, suppose that an allocation decision by S_1 leads to $\tau_i \underset{t}{\rightarrow} \tau_j$ and an allocation decision by S_2 leads to $\tau_j \underset{t}{\rightarrow} \tau_k$. This is a priority inversion since $(\tau_i \underset{t}{\leadsto} \tau_k) \wedge (\tau_i \succ \tau_k)$. Notice that neither S_1 nor S_2 maintains sufficient local information to detect or prevent this priority inversion.

We can enjoy the benefits of separate schedulers if, by analyzing each scheduler in isolation, freedom from priority inversions can be ensured. An obvious local criterion for correctness of a scheduler S is that S prevent priority inversions among tasks that have requested but not released resources from S. The two-scheduler example of the previous paragraph illustrates that this criterion by itself is not sufficient to ensure freedom from priority inversion—both S_1 and S_2 avoided such local priority inversions. We, therefore, now investigate useful conditions to ensure that avoiding local priority inversions is sufficient for avoiding all priority inversion.

Consider a system in which there is a set of independent schedulers and a single priority assignment that is known to all.[7] Define $\tau_i \overset{S}{\underset{t}{\rightarrow}} \tau_j$ to hold if

[7]The case where schedulers do not share a common priority assignment is discussed in

and only if $\tau_i \xrightarrow[t]{} \tau_j$ and scheduler S is delaying τ_i's request for some resource. The *local state* of a scheduler S can be characterized by a directed graph $G_S = (T, E_S)$ where $E_S = \{(u, v) \mid v \succ u \vee u \xrightarrow[t]{S} v\}$. Thus, G_S includes all of the priority edges of the composite system graph but only a subset of the waits-for edges. A *local priority inversion* exists for scheduler S if and only if G_S contains a π-cycle.

Since the composite system graph for a system with multiple schedulers is given by $G = (T, \bigcup_S E_S)$, a system contains a priority inversion at time t if and only if $\bigcup_S G_S$ contains a π-cycle. However, as illustrated in the two-scheduler example above, existence of a π-cycle in $\bigcup_S G_S$ does not imply a π-cycle in G_S for some scheduler S. The following theorem shows that this discrepancy is linked to specifying priority assignments with partial orders.

Theorem 2 *A system with multiple schedulers is free from priority inversion if the priority assignment is a total order and each scheduler avoids local priority inversions.*

Proof: By contradiction. Assume that the system has a priority inversion characterized by the π-cycle

$$\tau_i \Rightarrow \tau_j \longrightarrow \cdots \longrightarrow \tau_k \longrightarrow \tau_\ell \longrightarrow \cdots \longrightarrow \tau_i$$

in the composite system graph. For each $\tau_k \longrightarrow \tau_\ell$ in the π-cycle, we conclude $\tau_\ell \succ \tau_k$ because every pair of tasks is related by the priority assignment and no scheduler allows a local priority inversion. By the transitivity of $\succ$, we have $\tau_i \succ \tau_j$. But, from $\tau_i \Rightarrow \tau_j$ we conclude $\tau_j \succ \tau_i$ and obtain the contradiction that $\tau_i \succ \tau_j$ and $\tau_j \succ \tau_i$. □

Even in systems where the priority assignment is a partial order, it is possible to design schedulers that use only local information yet still manage to avoid all priority inversions. The strategy is for schedulers to be conservative and never permit a local configuration necessary for a π-cycle to develop. An example of such a strategy is given by the following theorem.

Theorem 3 *If a task τ_i is never allowed to wait when there exists τ_j such that $\tau_i \succ \tau_j$, then the system is guaranteed to be free from all priority inversions.*

Section 7.2.

Proof: The stated allocation policy prevents executions that lead to $\tau_j \Rightarrow \tau_i \rightarrow \tau$ for any τ. Since this is a necessary configuration for a π-cycle, the system cannot contain priority inversions. □

A symmetric policy, which prevents $\tau \rightarrow \tau_j \Rightarrow \tau_i$ configurations, also works. However, both policies may degenerate to "do not allocate any resource to any task," a policy that is not very useful. The simple system with three tasks τ_i, τ_j, τ_k and a priority assignment where $\tau_j \succ \tau_i$ and $\tau_k \succ \tau_i$ illustrates the problem with the first policy. Suppose all three tasks share a single processor. Task τ_i cannot be scheduled since this allocation decision would lead to $(\tau_j \underset{t}{\rightarrow} \tau_i) \wedge (\tau_j \succ \tau_i)$, a priority inversion. Task τ_j cannot be scheduled since it would lead to $\tau_k \underset{t}{\rightarrow} \tau_j$, which violates the scheduling policy that a task (τ_k) is never allowed to wait when there exists a task (τ_i) with lower priority. Finally, task τ_k cannot be scheduled for the symmetric argument. In other words, none of the three tasks can run even though two have incomparable priorities!

In many systems, the priority assignment is encoded by associating an integer priority $\Pi(\tau)$ with each task τ. If each task is assigned a unique priority, then the result is a total order and Theorem 2 implies that avoiding local priority inversions is sufficient for avoiding all priority inversions. However, constructing a total order from a partial order can require introduction of fictitious priority relations—avoiding priority inversions that involve these fictitious relations is unnecessary. Thus, we now consider the case where a unique priority is not assigned to each task, but Π does satisfy the following less-restrictive conditions, which define a partial order (as opposed to an irreflexive partial order).

P1. $\Pi(\tau) > \Pi(\tau')$ if and only if $\tau \succ \tau'$

P2. $\Pi(\tau) = \Pi(\tau')$ implies $\neg(\tau \succ \tau') \wedge \neg(\tau' \succ \tau)$.

Observe that for a given priority assignment, a mapping that satisfies P1 and P2 might require adding some fictitious priority relations, but would require adding fewer priority relations than if Π defined a total order. The following theorem asserts that even though Π defines a partial order, due to P1 and P2, avoiding local priority-inversions suffices to avoid all priority inversions.

Theorem 4 *If a task τ is only allowed to wait for a task τ' for which $\Pi(\tau) \leq \Pi(\tau')$ then the system is guaranteed to be free from all priority inversions.*

Proof: By contradiction. Assume a priority inversion characterized by the π-cycle

$$\tau_i \Rightarrow \tau_j \rightarrow \cdots \rightarrow \tau_k \rightarrow \tau_\ell \rightarrow \cdots \rightarrow \tau_i$$

in the composite system graph. From the allocation policy, $\tau_k \rightarrow \tau_\ell$ implies $\Pi(\tau_\ell) \geq \Pi(\tau_k)$ for any two tasks τ_k and τ_ℓ in the π-cycle. By transitivity, we have $\Pi(\tau_i) \geq \Pi(\tau_j)$. By the hypothesis, $\tau_j \succ \tau_i$ and thus by Γ1 we have $\Pi(\tau_j) > \Pi(\tau_i)$, a contradiction. □

Note that the problem illustrated above for the policy of Theorem 3 no longer exists. One possible mapping that corresponds to the priority assignment of the example is $\Pi(\tau_i) = 1, \Pi(\tau_j) = \Pi(\tau_k) = 2$. Since tasks τ_j and τ_k have equal priority levels, either one could be scheduled without risking a priority inversion in the global system state.

7 Discussion

The characterization of priority inversion given above is useful only to the extent that the formal model on which it is based correctly captures the relevant aspects of reality. We, therefore, now discuss the suitability of our model and the relaxation of certain of its restrictions.

7.1 Priority Assignments as Partial Orders

We have elected to formalize priority assignments using irreflexive partial orders rather than mappings from tasks to integers (as is done in many operating systems). This selection was made because irreflexive partial orders are more expressive. For example, there is no mapping of tasks to integers to state that a task τ_1 competes on an equal footing with both τ_2 and τ_3 but τ_3 has priority over τ_2. Such a mapping Ψ would have to satisfy $\Psi(\tau_1) = \Psi(\tau_2)$, $\Psi(\tau_1) = \Psi(\tau_3)$ and $\Psi(\tau_3) > \Psi(\tau_2)$. Also, using an irreflexive partial order avoids introducing fictitious priority relations, which, in turn, avoid fictitious priority inversions.

When irreflexive partial orders are used to specify priority assignments, there are two possible interpretations for the case where two tasks have incomparable priorities. One is that these tasks do not compete for resources; the other is that these tasks compete for resources but on an equal footing. In either case, if two tasks are incomparable then, by definition (see Section 2.1) each will be in the peer group of the other. At first, including in the peer group for τ tasks that do not compete for resources with τ might seem troubling. However, if two tasks do not compete for resources, then it doesn't matter whether one is in the peer group of the other—any analysis based on that peer group will be no more complicated by the presence of the non-competing task.

Using irreflexive partial orders to specify priority assignments does have drawbacks. As shown in Section 6, using individual schedulers that ensure freedom from local priority inversion does not by itself guarantee that a system will be free of priority inversions. However, Theorem 4 shows that if the priority assignment is restricted to one that could be represented by a mapping from tasks to integers, guaranteeing freedom from local priority inversions does guarantee freedom from all priority inversions. Thus, in systems with multiple, independent schedulers, there are advantages to employing the less-expressive formulation of priority assignment.

7.2 Static and Global Priority Assignment

One limitation of our model is the assumption of a single, static priority assignment. This rules out systems where a task's priority is a function of the system state. It also rules out systems with multiple independent schedulers that each assign different priorities to tasks. A time-varying or dynamic priority structure can be modeled as a sequence of priority assignments,

$$\succ_1, \succ_2, \ldots, \succ_t, \ldots$$

where $\succ_t$ is the priority assignment in effect at time t. The formal characterization of priority inversion in Section 3 remains valid with this extension, but the protocols of Sections 4 and 5 require modification. This is because if the priority assignment is not static, priority inversions can be caused simply by changing the priority assignment; with a static priority assignment, a priority inversion can only occur from an (ungranted) request operation. Avoiding priority inversions for a dynamic priority structure, therefore, requires that changes to the priority assignment be coupled with the waits-for relation.

Our definition of priority inversion does not work for the case where different schedulers can assign different priorities to tasks. To understand the problem, consider a system with two resources—a processor and a communications channel—and two tasks τ_1 and τ_2. Suppose the priority assignment $\succ_P$ used by the processor has $\tau_1 \succ_P \tau_2$ and the priority assignment $\succ_C$ used by the channel has $\tau_2 \succ_C \tau_1$. It is possible for $\tau_1 \underset{t}{\rightarrow} \tau_2$ due to an allocation decision by the communications channel and for $\tau_2 \underset{t}{\rightarrow} \tau_1$ due to an allocation decision by the processor. We believe that this scenario should not be considered a priority inversion because no task is being prevented from using a resource by a task that has a lower priority for the use of that resource. However, $(\tau_1 \succ \tau_2) \wedge (\tau_1 \underset{t}{\rightarrow} \tau_2)$ holds, which means there is a priority inversion according to the definition of Section 3.

7.3 Multiple-unit Resource Requests

Another limitation of our model is the requirement that tasks request individual resources sequentially. As a result of this limitation, we have not had to define what constitutes a priority inversion when a task can request multiple resources simultaneously. There is good reason for this omission—it is not clear what the correct definition should be. Consider a system consisting of two processors P_1 and P_2 and three tasks τ_1, τ_2 and τ_3. Further, suppose $\tau_1 \succ \tau_2$, τ_1 requires two processors, and τ_2 and τ_3 each require a single processor. It seems reasonable to claim that τ_2 executing on P_1 while P_2 is idle constitutes a priority inversion, since τ_2 holds a resource required by higher-priority task τ_1. What is not clear is whether τ_3 executing on P_1 while P_2 is idle should also be considered a priority inversion. On the one hand, no task holds resources required by a higher-priority task, suggesting that this should not be considered a priority inversion. On the other hand, if this is not considered a priority inversion then the seemingly harmless act of putting idle processor P_2 to work executing τ_2 should not be considered a priority inversion, either. Yet, τ_2 now holds a resource required by τ_1, implying that a priority inversion exists.

7.4 Bounded Priority Inversions

This paper has been concerned with avoiding priority inversions completely. However, it is sometimes acceptable for priority inversions to occur, provided they are bounded in duration. Suppose a task is being delayed by some lower priority task. If the duration of this delay has a known bound, then we could view the situation as if the delay were included in the fixed overhead associated with allocating the resource. So, by adding to the cost of a request operation any delay due to a priority inversion, an analysis using peer groups would remain valid (despite the priority inversion). The priority inheritance protocols in [10], for example, provide a way to bound priority inversions under certain circumstances; the protocols of [8] prevent priority inversions of unbounded duration.

Define a *D-bounded priority inversion* to be a priority inversion whose duration is not longer than D and an *unbounded priority inversion* to be one whose duration cannot be so bounded. Avoiding all but D-bounded priority inversions can be easier and less costly than avoiding all priority inversions.[8] For example, detection can be used to eliminate all but D-bounded priority inversions—tasks run their course and periodically a protocol is executed to detect and eliminate priority inversions that may have formed.[9] The frequency with which the detector runs determines the upper bound D on the duration of priority inversions.

In theory, it is easy to build a detector for priority inversions. The detector must construct the composite system graph from the information available to schedulers. In a distributed system with multiple independent schedulers, a distributed snapshot algorithm [3] would have to be employed for this purpose. Priority inversion detection is then simply a matter of checking for π-cycles in the composite system graph of a snapshot. Note, however, that a priority inversion might have vanished by the time it is detected because occurrence of a π-cycle is not a stable property [3]. Such "ghost" priority inversions do not cause problems, however, because they are not unbounded priority inversions.

[8]Analyzing a scheduling protocol to determine a bound D can be a hard problem, however [10].

[9]Elimination of the priority inversion requires either that resource allocations to tasks be preemptable or that task executions be abortable.

8 Conclusions

This paper gives a formal characterization of priority inversion and gives a set of sufficient conditions for its avoidance. Based on these conditions, we have been able to derive new protocols to avoid priority inversion. We have also been able to give conditions to avoid priority inversion in systems with multiple schedulers that do not communicate.

The existence of a theory characterizing priority inversions makes it possible to design both general and application-specific protocols to avoid priority inversions. The theory also permits the consequences of system design decisions to be better understood. For example, we were surprised to find that choosing between an irreflexive partial order and an integer mapping representation of priority assignment can be significant. We were also surprised that the definition of priority inversion is elusive for systems in which multiple resource requests are possible or in which independent schedulers use different priority assignments.

Acknowledgements We are grateful to Hideyuki Tokuda, John Lehoczky, and Lui Sha for their encouragement and reactions to an early draft of this paper. Nancy Lynch and Rob Cook have also provided helpful comments on this material.

References

[1] Robert Abbott and Hector Garcia-Molina. Scheduling Real-Time Transactions with Disk Resident Data. Technical Report CS-TR-207-89, Department of Computer Science, Princeton University, February 1989.

[2] Philip Bernstein, Vassos Hadzilacos and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[3] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions of Computer Systems*, vol. 3, no. 1, pp. 63-75.

[4] E. W. Dijkstra. *A Discipline of Programming.* Prentice–Hall, New Jersey, 1976.

[5] B. W. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, vol. 23, no. 2, pp. 105–117, February 1980.

[6] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment. *Journal of the ACM*, vol. 20, pp. 46–61, January 1973.

[7] K. Marzullo. Concurrency control for transactions with priority. Technical Report 89-996, Department of Computer Science, Cornell University, May 1989.

[8] B. Munch-Anderson and T. U. Zahle. Scheduling according to job priority with prevention of deadlock and permanent blocking. *Acta Informatica*, vol. 8, pp. 153–175, 1977.

[9] L. Sha. On Priority Scheduling and Priority Inversion. Technical Report, Department of Computer Science, Carnegie-Mellon University.

[10] L. Sha, R. Rajkumar and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Technical Report CMU-CS-87-181, Department of Computer Science, Carnegie-Mellon University.

[11] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Trans. on Database Systems* vol. 4, no. 2, pp. 180–209, June 1979.

[12] W. Zhao, K. Ramamritham and J. A. Stankovic. Preemptive Scheduling Under Time and Resource Constraints. *IEEE Transactions on Computers*, vol. C-36, pp. 949–960, August 1987.